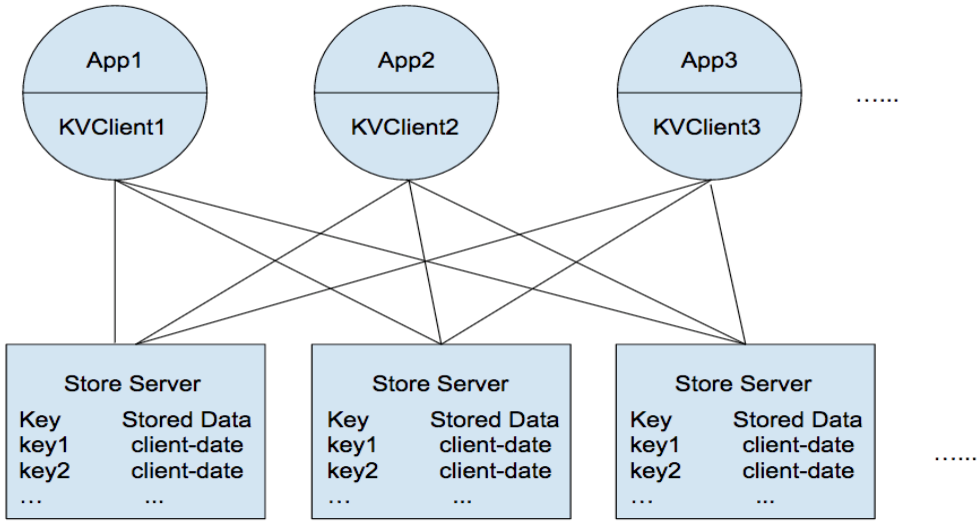
CPS 512 HW3 Transaction

Guotao Dong gd50@duke.edu **Meilong Pan** mp293@duke.edu

Introduction:

In this assignment, we extended KVStore and KVClient to implement serializable ACID transaction using Scala and Akka. There are RingServer tier, KVClient tier and KVServer tier in this service. The Application tier is the user of this transaction service. KVClient tier provides begin/commit/abort/read/write API to applications. KVServer is responsible for handling and guarantee the ACID properties of transactions, each data is routed to one shard of KVStore. We applied 2PL in handling locks acquisition and 2PC in deciding whether a transaction will be committed or aborted.



The figure above shows the architecture of our transaction service, the top layer is the application tier, which is ring servers in our case. The middle layer is KVClient tier, which functions as a library and binded with each ring server. The lower layer is Store server tier, which is used to store the data, handle lock acquire requests and complete transaction. Each data involved in one transaction is routed to one KVStore shard.

In following paragraphs, we will discuss how we design and implement our transaction service and show the test results.

Design and implementation

KVClient:

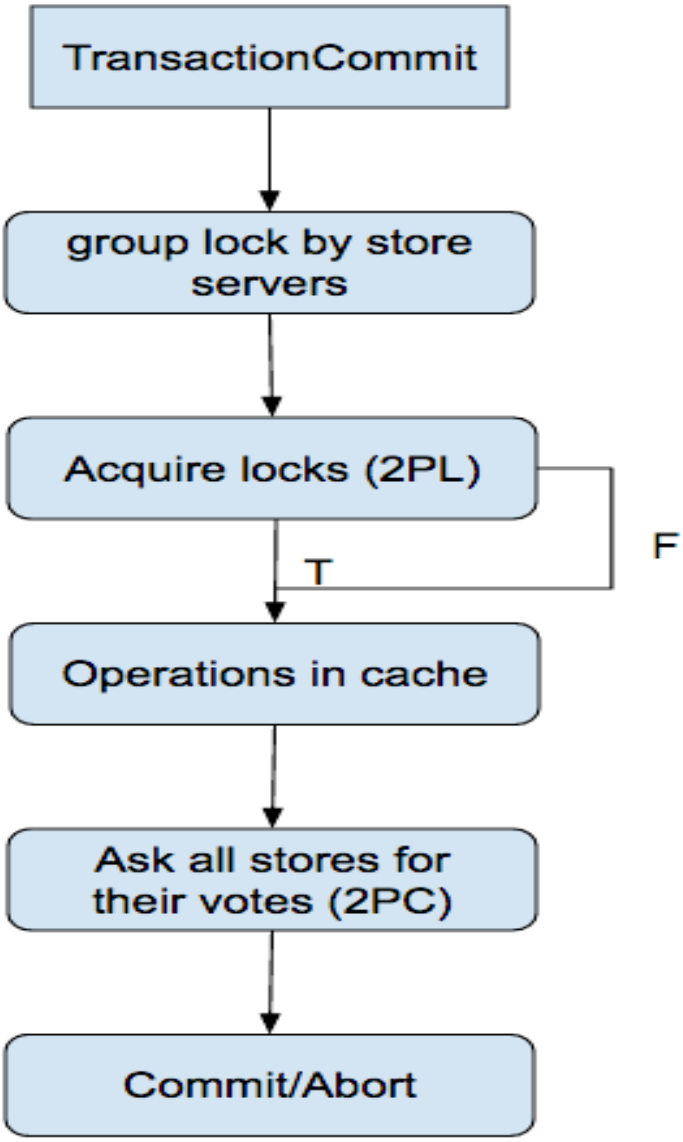
KVClient is the transaction library attached on each Ring Server which provides transaction APIs (begin/commit/abort/read/write) for application to use.

When TransactionBegin API is called, KVClient takes a snapshot of current cache for later recovery if current transaction aborts. KVClient will log all following TransactionRead and TransactionWrite.

When TransactionCommit API is called, KVClient first group all logs into store – keys mapping (so that lock request for locks on same store server can be completed with one request). Then, KVClient ask all involved store servers to lock the required data. This is where we applied 2 phase locking, we know all needed locks because we have logged all operations in previous steps. KVClient asks all involved store server to lock target keys. The lock request returns success only if all required locks are acquired. Otherwise KVClient will retry every 5 ms until successfully acquired all needed locks. We used exclusive lock in our implementation, locked data cannot be accessed by any other KVClient, in this way Isolation propertiy is guaranteed.

After acquired all locks, KVClient applies all logged read/write operations into their own cache. Then, we applied 2 phase commit to decide whether this transaction should commit or not. KVClient asks all involved store servers to vote, if all store server votes to commit, KVClient reply each store server that the transaction they just vote is commited, and each store server put the involved data into hard disk. Otherwise, KVClient informs each involved store server that the transaction is aborted and each store server not write involved data into hard disk, meanwhile, KVClient recovers its local cache to snapshot.

The flowchart below explains our TransactionCommit process.



KVServer:

KVServer is the component which handle data store, lock assignment, transaction votes. It provides messaging interface of GetLock, Unlock, Commit, CommitDecision. Each data is stored in one KVServer shard as a class named **StoreData**, which has fields of key, value, cacheOwner (client which holds this data in their cache), lockOwner (current lock owner of this data).

In **GetLock message interface**, KVStore checks to see if request lock can be assigned to request client. We divide this into two phases to avoid the rollback. In first

phase, KVStore check to see if all requested locks are available. If they are, then enters second phase. Otherwise, reply requester that their requested locks are not available. In second phase, KVStore assigns locks of requested data to requester.

In **Unlock message interface**, KVStore modifies the lockOwner field of specified data to be -1, which represents that the data is not holded by any one.

In **Commit message interface**, KVStore tries to write clients' operations into their own disk for durable purpose. In our application write failure is almost impossible, therefore we simulate it by generating a random number from 0 – 100, and determines if there is a write failure according to the number. If the number is from 0 – 95, it means write successfully and commit. If the number is from 95 – 100, it means there is a write failure and abort.

In **CommitDecision message interface**, KVStore do operations according to the KVClient's decision. If KVClient decides to commit this transaction, KVStore update the data's entry in store and informs all current data holders to clear their cache because this data is already rewritten. Then release resources taken by current KVClient because it is done with these resources in this transaction. Otherwise, release resources directly.

Partition Situation:

In this lab, we assume that KVStore is always fine and don't have to worry about the situation where KVStore is done. Therefore, we focus on solve the situation when KVClient is done.

In our application, KVClient fails before commit doesn't make any impact because they hold nothing from the KVStore, so we ignore this. Things get complicated when KVClient fails after commit and acquired all needed locks successfully. To solve this problem, we have to detect a partition firstly and then tackle it.

To detect a partition, we add heartbeat function in our KVClient. After acquiring locks successfully, KVClient heartbeats to KVStores every 5 ms to prove it is still alive. On KVStore side, if a data (e.g A) is currently held by one KVClient (client1), and another KVClient (client2) comes to lock the data (A), now we don't return false directly because it is possible that client1 is in a partition. What we do now is we check the latest heartbeat timestamp of the current holder of this data, if the diff of current system time and latest heartbeat timestamp is larger than the allowed interval, we think the current holder (client1) is in a partition.

Now we are able to detect a partitioned client using heartbeat, next we are going to solve the partition situation. On KVStore side, first we reclaim all resources held by

partitioned client and allow the new client to get their locks. Then we have to consider the situation when the partitioned client comes back, we have to let it know that it was in a partitioned state and its transaction is already aborted. What we do is we put a check statement at the very beginning of every message interface in KVStore. If the KVClient doesn't hold any resources in this KVStore (means it was in partition and comes back), we reply that KVClient and tell him that it was in a partition and its transaction is actually aborted. On Client side, when getting this "Partition" reply from the KVStore, the KVClient recovers its cache back to snapshot and tell other KVStore shards (have data held by this client but doesn't know this client was in partition yet) that this client was in partition, all resources held by it can be released now.

Test Results

In our design, each write operation on data increment the data by 1.

Deadlock test

```
/************  deadlock check ************/
//for (i <- 0 until 100) {
  servers(0) ! TransactionBegin()
  servers(1) ! TransactionBegin()
  servers(0) ! TransactionWrite(0)
  servers(1) ! TransactionWrite(1)
  servers(0) ! TransactionWrite(1)
  servers(1) ! TransactionWrite(0)
  servers(0) ! TransactionCommit()
  servers(1) ! TransactionCommit()
//}
/*****************************************/
```

Figure 1

First we tested the situation where KVClient0 and KVClient1 contend for key0 and key1 as shown in Figure 1. In this test we run one iteration here because we want to show the log and prove that our transaction has right behavior.

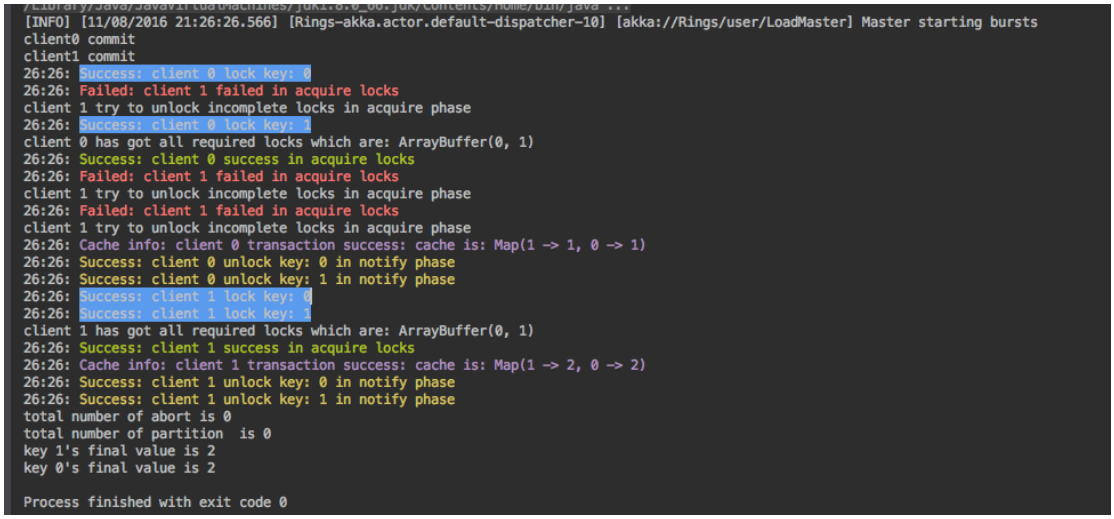


Figure 2

Figure 2 shows the log of this test. From the log we can see that client 0 get lock for data 0 firstly, then client 1 reports that it fails in acquire locks (which accords to the behavior of 2PL, “all or none”), then client 0 get lock for data 1 and reports that it has got all locks and its transaction is successful. After Client 0 release all locks he has, client 1 successfully get locks for data 0 and 1 and finish its transaction. This process meets our expectation and transaction server deals with key conflicting in a proper manner.

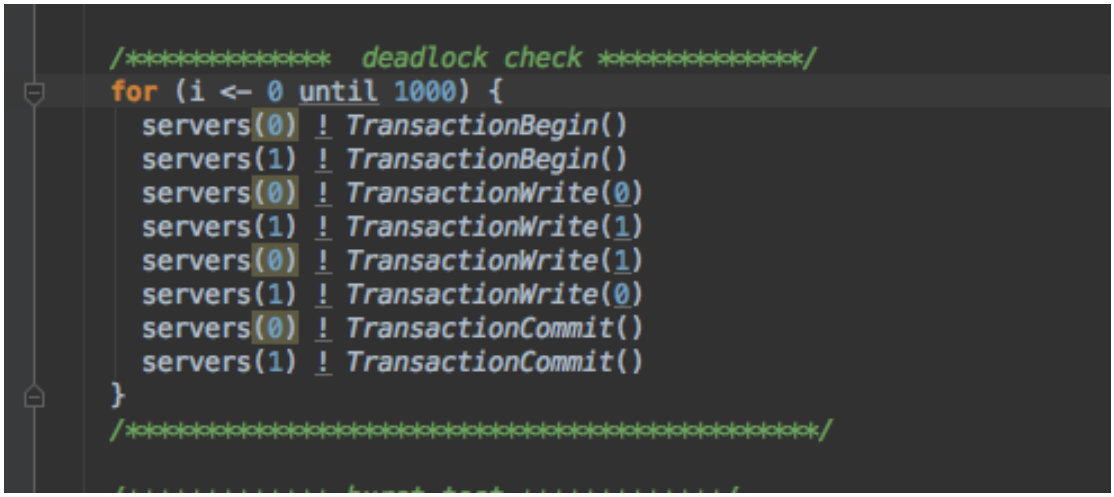


Figure3

Then in figure 3 we repeat the lock acquire pattern for 1000 times to check the result.

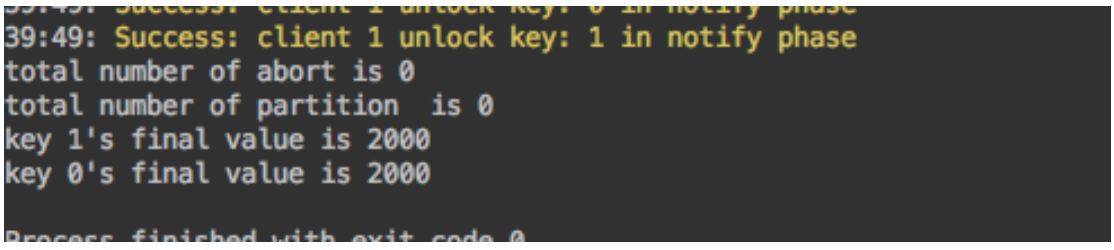


Figure 4

In Figure 4 we can see the final result of data 1 and data 2 are both 2000 which is right.

Read/Write API Test

In this section we test whether client's Read/Write API functions as expected.

```
/************  API Test ************/
//    for (i <- 0 until 1000) {
      servers(0) ! TransactionBegin()
      servers(0) ! TransactionWrite(0)
      servers(0) ! TransactionWrite(1)
      servers(0) ! TransactionWrite(2)
      servers(0) ! TransactionCommit()
      Thread.sleep(10)
      servers(1) ! TransactionBegin()
      servers(1) ! TransactionRead(0)
      servers(1) ! TransactionRead(1)
      servers(1) ! TransactionRead(2)
      servers(1) ! TransactionCommit()
      Thread.sleep(10)
      servers(1) ! TransactionBegin()
      servers(1) ! TransactionWrite(0)
      servers(1) ! TransactionWrite(1)
      servers(1) ! TransactionWrite(2)
      servers(1) ! TransactionCommit()
      Thread.sleep(10)
      servers(0) ! TransactionBegin()
      servers(0) ! TransactionRead(0)
      servers(0) ! TransactionRead(1)
      servers(0) ! TransactionRead(2)
      servers(0) ! TransactionCommit()
//    }
/*************************************************/
```

Figure 5

As usual we put a normal use case to see the log and we tested as Figure 5, in this test **client 0 write** data 0, 1, 2 in one transaction, then **client 1 read** data 0, 1, 2 in next transaction, then **client 1 write** data 0, 1, 2 in next transaction and finally **client 0 read** data 0, 1, 2 in last transaction. This test exposes both read and write

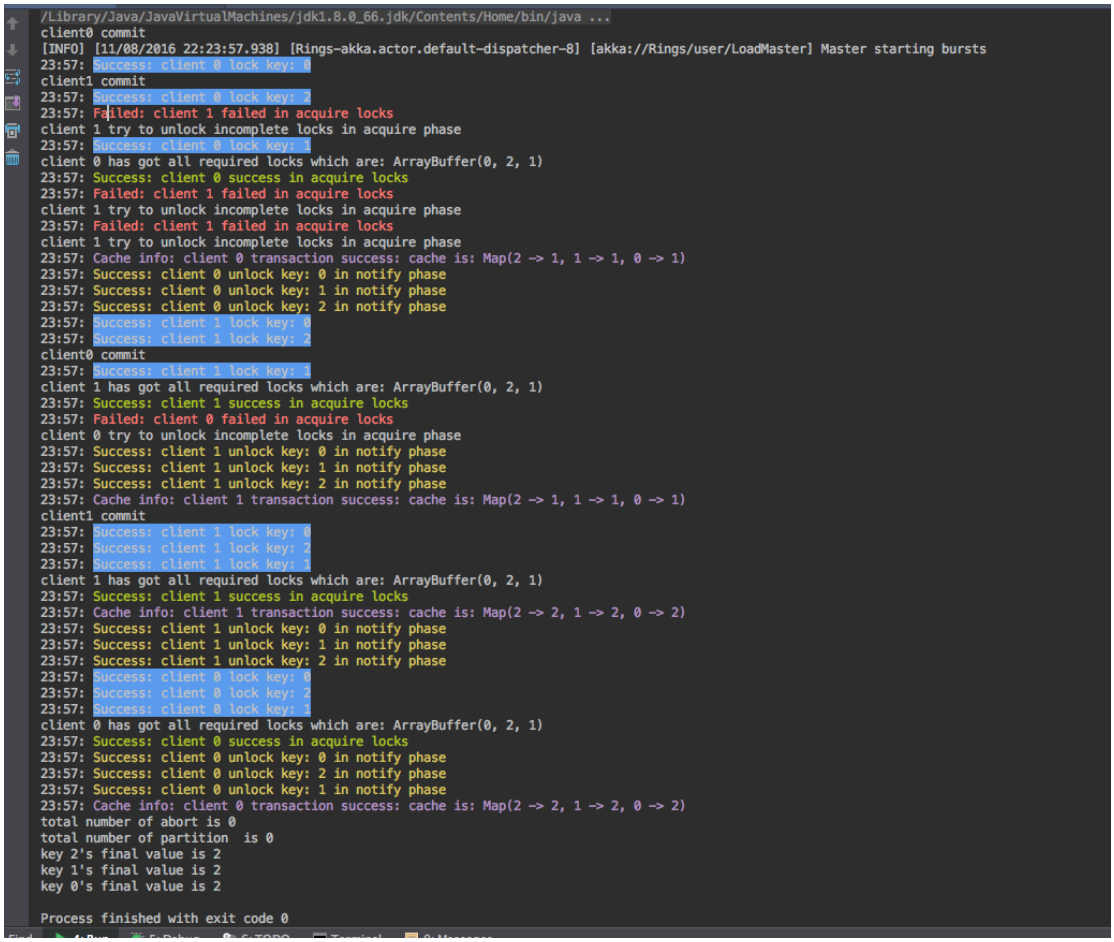


Figure 6

Figure 6 shows the log of this test, we can see from the test that client 0's write transaction is completed first, then client 1's read transaction is completed and now client 1's cache has the most updated value of data 0, data 1 and data 2. Then client 1's write transaction completed and finally client 0's read transaction. We can see from client 0's cache that finally he has the most updated data.

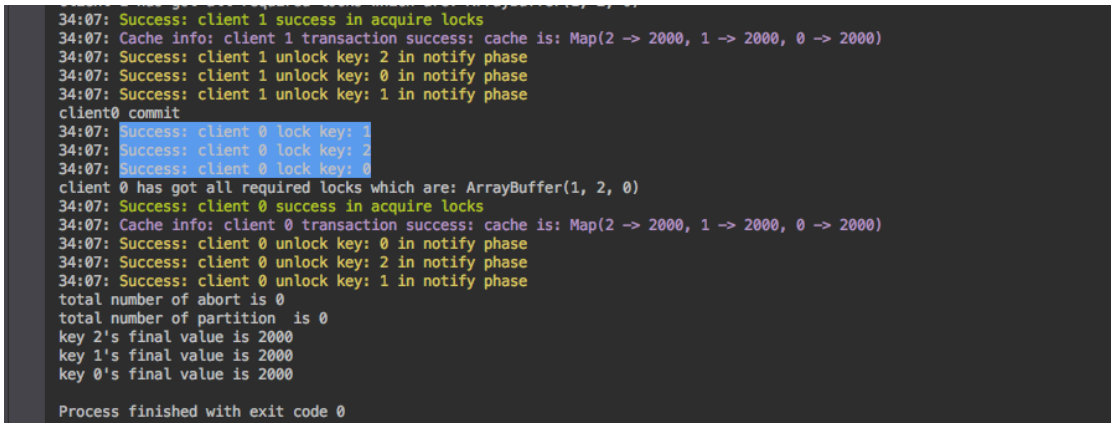


Figure 7

Then we iterate this test pattern for 1000 iterations, and the final result of data 0, data 1 and data 2 in their store server is 2000, which is the right answer. And the value of

these data in client 0 and client 1's cache is also 2000, which is the right answer.

Write Test

```
/************* burst test *************/
for (i <- 0 until burstSize) {
  // repeat time for each client
  for (j <- 0 until numClients) {
    // all clients write key 1 - 20 in underterministic order
    servers(j) ! TransactionBegin()
    for (k <- 0 until 10) {
      servers(j) ! TransactionWrite(k)
    }
    servers(j) ! TransactionCommit()
  }
}
/*****************************************/
```

Figure 8

In this test, we burst write operations to see if the result of large amount of write is right. Our test case is as shown in Figure 8. In this test case, we have 100 iteration, in each iteration, we have 10 KVClients writing the same data (data 0 to data 9).

```
total number of abort is 0
total number of partition  is 0
key 8's final value is 1000
key 2's final value is 1000
key 5's final value is 1000
key 4's final value is 1000
key 7's final value is 1000
key 1's final value is 1000
key 9's final value is 1000
key 3's final value is 1000
key 6's final value is 1000
key 0's final value is 1000
```

Figure 9

Figure 9 shows the result of this test, the final value of each data is 1000, which is the correct result.

Partition Test

In this test, we simulate the situation which one KVClient fails/partitioned after getting votes from all store servers, and data is acquired by another KVClient at the same time.

```
/*************  client 0 in partition ******/
    servers(0) ! TransactionBegin()
    servers(0) ! TransactionWrite(1)
    servers(0) ! TransactionCommit()
    Thread.sleep(1)
    servers(1) ! TransactionBegin()
    servers(1) ! TransactionWrite(1)
    servers(1) ! TransactionCommit()
 /*****************************************************/
```

Figure 10

As is shown in Figure 10, the test code is simple because in this test our main goal is to prove that the partition behavior meets our expectation.

```
/Library/Java/JavaVirtualMachines/jdk1.8.0_66.jdk/Contents/Home/bin/java ...
[INFO] [11/08/2016 23:21:13.683] [Rings-akka.actor.default-dispatcher-11] [akka://Rings/user/LoadMaster] Master starting bursts
client1 commit
client0 commit
21:13: Success: client 0 lock key: 1
21:13: Failed: client 1 failed in acquire locks
client 1 try to unlock incomplete locks in acquire phase
client 0 has got all required locks which are: ArrayBuffer(1)
21:13: Success: client 0 success in acquire locks
client 0 is partitioned after getting votes
21:13: Failed: client 1 failed in acquire locks
client 1 try to unlock incomplete locks in acquire phase
detect client 0 is partitioned when client 1 acquiring lock for key 1
21:13: Success: client 0 key: 1 reclaimed because it is partitioned
21:13: Success: client 1 lock key: 1
client 1 has got all required locks which are: ArrayBuffer(1)
client 0 recovers and send another heartbeat to me
21:13: Success: client 1 success in acquire locks
21:13: Cache info: client 1 transaction success: cache is: Map(1 -> 1)
21:13: Success: client 1 unlock key: 1 in notify phase
client 0 recovers and send another heartbeat to me
client 0 recovers and send another heartbeat to me
client 0 recovers after partition
21:13: Cache info: client 0 transaction success: cache is: Map(1 -> 1)
client 0 is notified as partitioned before
client 0 is cleaned
21:13: Cache info: client 0 transaction failure (partitioned): cache is: Map()
client 0 is notified as partitioned before
client 0 is cleaned
21:13: Cache info: client 0 transaction failure (partitioned): cache is: Map()
client 0 is notified as partitioned before
client 0 is cleaned
21:13: Cache info: client 0 transaction failure (partitioned): cache is: Map()
client 0 is notified as partitioned before
client 0 is cleaned
21:13: Cache info: client 0 transaction failure (partitioned): cache is: Map()
total number of abort is 0
total number of partition  is 1
key 1's final value is 1

Process finished with exit code 0
```

Figure 11

Figure 11 shows the log when we simulate a 30 ms partition after Client 0 get all required votes from all store servers. We can see from the log that after partition, client 0's lock on data 1 is taken by Client 1. And when client 0 recovers form the partition, it starts heartbeat again, and notified as partitioned before. The log shows that the behavior is correct.